Assignment Two

Brian Lynch

brian.lynch2@marist.edu

October 2, 2020

1 Selection Sort

1.1 Imports and Variables

The Selection class imports the ArrayList class from the java.util package. This is used to store and sort the values from *allMagicItems*. The field variable, Integer *comparisons*, will count how many value comparisons are required to sort the list.

1.2 Sort

The sort(ArrayList<String> allMagicItems) method functions in a manner that requires two loop functions. The first loop on [Line 7] incrementally ascends through the *allMagicItems* list. On [Line 9 - Line 13], the remaining values in the list (any value including or after *index*) are compared to find the smallest value. The variable *swapIndex* is assigned to hold this value. On [Line 8], *swapIndex* was preassigned to the value of *index*, to account for the cases that *index* holds the smallest value out of the remainder of the list. [On Line 17 - Line 19], this smallest value is swapped with *index*. **REWORD**

2 Insertion Sort

2.1 Imports and Variables

The Selection class imports the ArrayList class from the java.util package. This is used to store and sort the values from *allMagicItems*. The field variable, Integer *comparisons*, will count how many value comparisons are required to sort the list.

2.2 Sort

The sort(ArrayList<String> allMagicItems) method, similar to Selection Sort, functions in a manner that requires two loop functions. The first loop on [Line 7] incrementally ascends through the *allMagicItems* list. The nested loop on [Line 8], starts at the index from the previous loop, and incrementally descends through the *allMagicItems* list. On [Line 11], if the value of the index of the descending loop, represented

by the value at *tempIndex*, is less than the preceding value in the list, the two values will be swapped [Line 13 - Line 15]. **REWORD ** If the value is not less than its predecessor, the descending loop will break [Line 18] and the ascending loop will increment to the next value in the list.

3 Merge Sort

3.1 Imports and Variables

The Selection class imports the ArrayList class from the java.util package. This is used to store and sort the values from *allMagicItems*. The field variable, Integer *comparisons*, will count how many value comparisons are required to sort the list.

3.2 Sort

The sort(ArrayList<String> allMagicItems) method sorts, in a generalized sense, by recursively dividing the list in half until each value is in its own unique list. Then, the values will merge back together in ascending order from lowest to highest value. The method starts off with a condition that tests if the list still need to be divided, and is greater than size one [Line 6]. On [Line 8 - Line 14], the list *allMagicItems* is divided in half, and the bottom and top halves recurse separately. After the lists can no longer be divided, the rest of the code, starting on [Line 17], will execute. On [Line 17 - Line 40], two lists are compared, and are sorting in ascending order. On [Line 22 - Line 29], the if statement is executed if the lists *bottomList* and *topList* both still have values to be placed. On [Line 24 - Line 29], the lesser value between the two lists replaces the value in the list *allMagicItems*. On [Line 31 - Line 38], if any values in either the *bottomList* or *topList* remain, but the other list is completely, the remaining values will be dumped into *allMagicItems*.

4 Quick Sort

4.1 Imports and Variables

The Selection class imports the ArrayList and Random classes from the java.util package. The ArrayList class is used to store and sort the values from *allMagicItems*. The Random class to used to find a random value from *allMagicItems*, which will be used as the "pivot" value. The field variable, Integer *comparisons*, will count how many value comparisons are required to sort the list.

4.2 Sort

The sort(ArrayList<String> allMagicItems) method begins with a condition that if the list *allMagicItems* is greater than size one, it will execute. On [Line 10], the partition method is called, and returns the index of the "pivot" value. The "pivot" value is used to divide and sort the list. On [Line 13 - Line 16], the list will be divided as values less than the "pivot" value are stored in *bottomList*, and values greater than the "pivot" value are stored in *topList*. These two lists are then recursed separately. When the compiler reaches [Line 19], which is the merge phase, *allMagicItems* will have been completely divided and in already in ascending order. On [Line 19 - Line 27], *allMagicItems* is reformed by replacing its values with *bottomList*, then the "pivot" value, and then *topList*.

4.3 Partition

The purpose of the partition(ArrayList<String> allMagicItems) method is to order the list around the "pivot" value. On [Line 34 - Line 36], a random value is selected from *allMagicItems* and is assigned

to *pivot*. On [Line 39 - Line 40], the *pivot* value is swapped with the last value in the list. This swap will assist greatly in the functionality and simplicity of the comparison phase. On [Line 43 - Line 50], the values in $allMagicItems$ are compared to *pivot*. If the value is less than *pivot*, then the value is swapped with the value at *index*, which ascends from the starting from the beginning of the list. After all values less than *pivot* have been placed, *pivot* is swapped with the value at *index*, which is the value proceeding the most recently placed value. At this time, $allMagicItems$ is arranged with all values at indexes (0...pivot) are less than all values at indexes (pivot+1...allMagicItems.size()-1).

5 Main

5.1 Imports and Variable

The Main class uses four imports. From the java.io package, the File and FileNotFoundException classes are used. From the java.util package, the Scanner and ArrayList classes are used.

5.2 File and Array

The main method starts with a try-catch statement that detects if there are any issues reading the "magicitems.txt" file. On [Line 10 - Line 16], the "magicitems.txt" file is retrieved and read. Then, the ArrayList $allMagicItems$ stores each line of the file indvidually.

5.3 Sorts

On [Line 20 - 45], the different sorts are executed. Each sort call operates in the same manner. To keep the integrity of the control data ($allMagicItems$), an identical copy of the list data was created and used for each of the different sorts. Then, the class objects are created, and their sort methods are called. Each sort, aside from Quick Sort, had consistent results in the number of comparisons. The reason Quick Sort had inconsistent results was due to the fact that the "pivot" value was randomized. A "pivot" value that is closer to the list's median value will be more effective than a value closer to an edge of the list. On [Line 49 - Line 57], to get more accurate results, I found the average number of comparisons using a data set of 100,000 Quick Sort objects. An average and more stable number is found, being 6780 comparisons.

Food For Thought: On my Average Quick Sort variation, I tried to take the average of a data set of 1,000,000 Quick objects instead of 100,000. The output ended up being -1810. I got this result multiple times. What is causing this to happen?

6 Results

For the run times, Selection sort has a best and worst Run Time of $O(n^2)$. More accurately, before simplification, the value is $n(n-1)/2$. This accounts for a loop iterating through the list n times, and then iterating again $n-1$ with a nested loop. When n is substituted with 666, representing the amount of items in "magicitems.txt", the result is 221445, equivalent to the number of comparisons. Insertion sort has a range with its best time being $O(n)$ and worst time $O(n^2)$. Insertion sort will near the value $O(n)$ depending upon how close to sorted the list already is. If the list is already pre-sorted, the loop will iterate through the list n times. The best and worse case for Merge sort is $O(n * lg(n))$. This is due to the fact that consistently, Merge sort will always divide the list in half and then recurse each half. Since it's division is consistent, the Run Times are stable. Before simplification, the value is actually $n * lg(n) + n$. The equation comes from the fact that Merge sort uses recursion. When comparing against a recursion tree, the run time can be found by

multiplying n (the size of the array) by $lg(n) + 1$ (the levels of recursion). Quick Sort has similar run times to Merge sort, however Quick Sort is less stable. Its best time is $O(n * lg(n))$ and its worse time is $O(n^2)$. This is due to the fact that Quick Sort uses a random "pivot" value from the list. If, for example, on every occasion the "pivot" value was the median value in the current list, the run time would be $O(n * lg(n))$. If, for another example, every "pivot" value was an edge value from the list, the run time would be $O(n^2)$.

Sort	Comparisons	Run Time Best	Run Time Worst
Selection	221445	$O(n^2)$	$O(n^2)$
Insertion	114976	$O(n)$	$O(n^2)$
Merge	5413	$O(n * lg(n))$	$O(n * lg(n))$
Quick Av.	6780	$O(n * lg(n))$	$O(n^2)$

7 References

7.1 Selection Sort

```
import java.util.ArrayList;

public class Selection {
    int comparisons = 0;
    void sort(ArrayList<String> allMagicItems){
        for(int index = 0; index<allMagicItems.size()-1;index++){
            //set swapIndex to index. if swapIndex does not change value, index will be
                swapped with itself.
            int swapIndex = index;
            for(int i = index+1; i<allMagicItems.size();i++){
                comparisons++;
                //find smallest value in remainder of array, and set it to swapIndex.
                if(allMagicItems.get(i).compareTo(allMagicItems.get(swapIndex))<0){
                    swapIndex = i;
                }
            }
            //swap lowest value from remainder of list with index
            String tempString = allMagicItems.get(swapIndex);
            allMagicItems.set(swapIndex,allMagicItems.get(index));
            allMagicItems.set(index,tempString);
        }
    }
}
```

7.2 Insertion Sort

```
import java.util.ArrayList;

public class Insertion {
    int comparisons = 0;
    void sort(ArrayList<String> allMagicItems){
        //increments ascending through list, and compares descending through the list
        for(int index = 1; index<allMagicItems.size(); index++){
            for(int tempIndex = index; tempIndex>0;tempIndex--){
                //compares item to previous item in list
                comparisons++;
                if(allMagicItems.get(tempIndex).compareTo(allMagicItems.get(tempIndex-1))<0)
                    {
                    //swap item with previous item in list
                    String tempItem = allMagicItems.get(tempIndex);
                    allMagicItems.set(tempIndex,allMagicItems.get(tempIndex-1));
                    allMagicItems.set(tempIndex-1,tempItem);
                } else{
                    break;
                }
            }
        }
    }
}
```

7.3 Merge Sort

```
import java.util.ArrayList;

public class Merge {
    int comparisons = 0;
    void sort(ArrayList<String> allMagicItems) {
        if (allMagicItems.size() > 1) {
            //split list into 2 separate lists
            int midPoint = allMagicItems.size() / 2;
            ArrayList<String> bottomList = new ArrayList<String>(allMagicItems.subList(0, midPoint));
            ArrayList<String> topList = new ArrayList<String>(allMagicItems.subList(midPoint, allMagicItems.size()));

            //recurse bottom and top lists
            sort(bottomList);
            sort(topList);

            //merge the bottom and top lists
            int bottomIndex = 0;
            int topIndex = 0;
            for(int index = 0; index<allMagicItems.size(); index++){
                //tests if bottomIndex has additional testable values, topIndex has additional testable values, or both
                if(bottomIndex<bottomList.size() && topIndex<topList.size()){
                    //tests lower value between bottom or top list and places into main array
                    comparisons++;
                    if(bottomList.get(bottomIndex).compareTo(topList.get(topIndex))<0){
                        allMagicItems.set(index,bottomList.get(bottomIndex));
                        bottomIndex++;
                    } else{
                        allMagicItems.set(index,topList.get(topIndex));
                        topIndex++;
                    }
                } else if(bottomIndex<bottomList.size()){
                    //if executes, only items in bottom list still need to be placed. places remaining values
                    allMagicItems.set(index,bottomList.get(bottomIndex));
                    bottomIndex++;
                } else if(topIndex<topList.size()){
                    //if executes, only items in top list still need to be placed. places remaining values
                    allMagicItems.set(index,topList.get(topIndex));
                    topIndex++;
                }
            }
        }
    }
}
```

,

7.4 Quick Sort

```
import java.util.ArrayList;
import java.util.Random;

public class Quick {
    int comparisons = 0;
    void sort(ArrayList<String> allMagicItems) {
        if (allMagicItems.size() > 1 ) {
            //find the pivot index, and place all values that are less than the pivot before
                 the index,
            //and all values greater than the pivot after the index
            int pivotIndex = partition(allMagicItems);

            //create and sort lists of values less than the pivot, and values greater than
                the pivot
            ArrayList<String> bottomList = new ArrayList<String>(allMagicItems.subList(0,
                pivotIndex));
            ArrayList<String> topList = new ArrayList<String>(allMagicItems.subList(
                pivotIndex + 1, allMagicItems.size()));
            sort(bottomList);
            sort(topList);

            //merge the bottom list, pivot, and top list back together.
            int bottomIndex = 0;
            int topIndex = 0;
            for(int mainIndex = 0; mainIndex<allMagicItems.size();mainIndex++){
                if(mainIndex<pivotIndex){
                    allMagicItems.set(mainIndex,bottomList.get(bottomIndex));
                    bottomIndex++;
                } else if(mainIndex>pivotIndex){
                    allMagicItems.set(mainIndex, topList.get(topIndex));
                    topIndex++;
                }
            }
        }
    }
    Integer partition(ArrayList<String> allMagicItems){
        //get random value from list
        Random r = new Random();
        int pivotIndex = r.nextInt(allMagicItems.size());
        String pivot = allMagicItems.get(pivotIndex);

        //swap pivot with last value in list
        allMagicItems.set(pivotIndex,allMagicItems.get(allMagicItems.size()-1));
        allMagicItems.set(allMagicItems.size()-1,pivot);

        //if value is less than pivot, place at next index at start of list
        int index = 0;
        for(int j = index; j<allMagicItems.size()-1; j++){
            comparisons++;
            if(allMagicItems.get(j).compareTo(pivot)<0){
                String tempString = allMagicItems.get(j);
                allMagicItems.set(j,allMagicItems.get(index));
                allMagicItems.set(index,tempString);
                index++;
            }
        }
        //swap pivot with value at index. This places all values (start...pivot)<(pivot+1...
            end)
        allMagicItems.set(allMagicItems.size()-1,allMagicItems.get(index));
        allMagicItems.set(index,pivot);

        //return index of pivot value
        return index;
```

```
59        }
60    }
```

7.5 Main

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
import java.util.ArrayList;

public class Main {
    public static void main(String[] args){
        try {
            //read and retrieve file
            File itemFile = new File("magicitems.txt");
            Scanner readFile = new Scanner(itemFile);

            //initialize and develop arraylist of items
            ArrayList<String> allMagicItems = new ArrayList<String>();
            while(readFile.hasNextLine()){
                allMagicItems.add(readFile.nextLine().toUpperCase().replace(" ",""));
            }

            //* Selection Sort
            ArrayList<String> selectionArray = new ArrayList<String>(allMagicItems);
            Selection s = new Selection();
            s.sort(selectionArray);
            System.out.println("Selection: " + s.comparisons);
            //*/

            //* Insertion Sort
            ArrayList<String> insertionArray = new ArrayList<String>(allMagicItems);
            Insertion i = new Insertion();
            i.sort(insertionArray);
            System.out.println("Insertion: " + i.comparisons);
            //*/

            //* Merge Sort
            ArrayList<String> mergeArray = new ArrayList<String>(allMagicItems);
            Merge m = new Merge();
            m.sort(mergeArray);
            System.out.println("Merge: " + m.comparisons);
            //*/

            //* Quick Sort
            ArrayList<String> quickArray = new ArrayList<String>(allMagicItems);
            Quick q = new Quick();
            q.sort(quickArray);
            System.out.println("Quick: " + q.comparisons);
            //*/

            /* EXTRA Average Quick Sort
            int sum = 0;
            int iterations = 100000;
            for(int count = 0; count<iterations; count++){
                ArrayList<String> averageQArray = new ArrayList<String>(allMagicItems);
                Quick averageQ = new Quick();
                averageQ.sort(averageQArray);
                sum += averageQ.comparisons;
            }
            System.out.println("Quick Sort Average: " + sum/iterations);
            //*/

        //prints error message if file is not found
        } catch(FileNotFoundException a){
            System.out.println("Error: File not found.");
        }
    }
```

```
64 }
```